(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
24 July 2003 (24.07.2003)

PCT

(10) International Publication Number
WO 03/060705 A2

(51) International Patent Classification[7]: G06F 9/40

(21) International Application Number: PCT/US03/00876

(22) International Filing Date: 10 January 2003 (10.01.2003)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
60/347,773 11 January 2002 (11.01.2002) US
60/373,359 17 April 2002 (17.04.2002) US

(71) Applicant: SUN MICROSYSTEMS, INC. [US/US]; 4150 Network Circle, Santa Clara, CA 95054 (US).

(72) Inventors: MOIR, Mark, S.; 108 Liberty Road, #2, Somerville, MA 02144 (US). LUCHANGCO, Victor; 29A Lewis Ave., Arlington, MA 02474 (US). HERLIHY, Maurice; 18 Russell St, Brookline, MA 02446 (US).

(74) Agents: ZAGORIN O'BRIEN & GRAHAM LLP, O'BRIEN, DAVID, W. ET AL. et al.; 401 West 15th Street, Suite 870, Austin, TX 78701 (US).

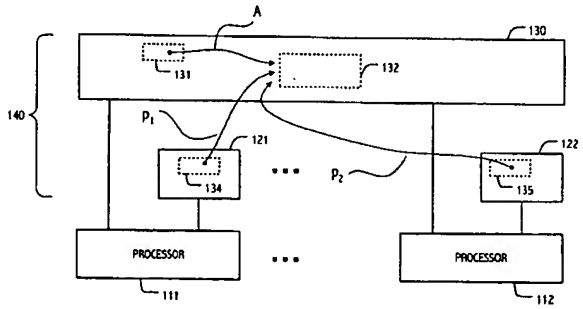
(81) Designated States *(national)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, OM, PH, PL, PT, RO, RU, SC, SD, SE, SG, SK, SL, TJ, TM, TN, TR, TT, TZ, UA, UG, UZ, VC, VN, YU, ZA, ZM, ZW.

(84) Designated States *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, SK, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:
— *without international search report and to be republished upon receipt of that report*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

(54) Title: LOCK-FREE IMPLEMENTATION OF DYNAMIC-SIZED SHARED DATA STRUCTURE



(57) Abstract: Solutions to a value recycling problem that we define herein facilitate implementations of computer programs that may execute as multithreaded computations in multiprocessor computers, as well as implementations of related shared data structures. Some exploitations of the techniques described herein allow non-blocking, shared data structures to be implemented using standard dynamic allocation mechanisms (such as malloc and free). A variety of solutions to the proposed value recycling problem may be implemented. A class of general solutions to value recycling is described in the context of an illustration we call the Repeat Offender Problem (ROP), including illustrative Application Program Interfaces (APIs) defined in terms of the ROP terminology. Furthermore, specific solutions, implementations and algorithm, including a Pass-The-Buck (PTB) implementation are also described. Solutions to the value recycling problem can be applied in a variety of ways to implement dynamic-sized data structures. For example, specific solutions are illustrated in the context of particular shared data structures and algorithms, e.g., a lock-free FIFO queue for which we demonstrate true dynamic sizing. In some cases, data structure implementations may be directly coded in ways that exploit the value recycling techniques described herein. In others, a single-word lock-free reference counting (SLFRC) technique (which builds on a value recycling solution) may be employed to transform, in a straight-forward manner, many lock-free data structure implementations that assume garbage collection (i.e., which do not explicitly free memory) into dynamic-sized data structures.

WO 03/060705 A2

LOCK-FREE IMPLEMENTATION OF DYNAMIC-SIZED SHARED DATA STRUCTURE

Technical Field

The present invention relates generally to coordination amongst execution sequences in a multiprocessor computer, and more particularly, to structures and techniques for facilitating implementations of concurrent data structures and/or programs.

Background Art

Management of dynamically allocated storage presents significant coordination challenges for multithreaded computations. One clear, but important, challenge is to avoid dereferencing pointers to storage that has been freed (typically by operation of another thread). Similarly, it is important to avoid modifying portions of a memory block that has been deallocated from a shared data structure (e.g., a node removed from a list by operation of another thread). These and other challenges are generally well recognized in the art.

A common coordination approach that addresses at least some of these challenges is to augment values in objects with version numbers or tags, and to access such values only through the use of Compare-And-Swap (CAS) instructions, such that if a CAS executes on an object after it has been deallocated, the value of the version number or tag will ensure that the CAS fails. *See e.g.*, M. Michael & M. Scott, *Nonblocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors*, *Journal of Parallel and Distributed Computing*, 51(1):1-26, 1998. In such cases, the version number or tag is carried with the object through deallocation and reallocation, which is usually achieved through the use of explicit memory pools. Unfortunately, this approach has resulted in implementations that cannot free memory that is no longer required.

Valois proposed another approach, in which the memory allocator maintains reference counts for objects in order to determine when they can be freed. *See* J. Valois, *Lock-free Linked Lists Using Compare-and-Swap*, in *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 214-22, 1995. Valois' approach allows the reference count of an object to be accessed even after the object has been released to the memory allocator. This behavior restricts what the memory allocator can do with released objects. For example, the released objects cannot be coalesced. Thus, the disadvantages of maintaining explicit memory pools are shared by Valois' approach. Furthermore, application designers sometimes need to switch between different memory allocation implementations for performance or other reasons. Valois' approach requires the memory allocator to support certain nonstandard functionality, and therefore precludes this possibility. Finally, the space overhead for per-object reference counts may be prohibitive. We have proposed another approach that does allow memory allocators to be interchanged, but depends on double compare-and-swap (DCAS), which is not widely supported. *See e.g.*, commonly-owned, co-pending U.S. Application No. 09/837,671, filed April 18, 2001, entitled "Lock-Free Reference Counting," and naming David L. Detlefs, Paul A. Martin, Mark S. Moir and Guy L. Steele Jr. as inventors.

Interestingly, the work that may come closest to meeting the goal of providing support for explicit non-blocking memory management that depends only on standard hardware and system support predates the work discussed above by almost a decade. Treiber proposed a technique called *obligation passing*. *See* R. Treiber, *Systems Programming: Coping with Parallelism*, Technical Report RJ5118, IBM Almaden Research Center, 1986. The instance of this technique for which Treiber presents specific details is in the implementation of a lock-free linked list supporting search, insert, and delete operations. This implementation allows freed nodes to be returned to the memory allocator through standard interfaces and without requiring special functionality of the memory allocator. However, it employs a "use counter" such that memory is reclaimed only by the "last" thread to access the linked list in any period. As a result, this implementation can be prevented from ever recovering any memory by a failed thread (which defeats one of the main purposes of using lock-free implementations). Another disadvantage of this implementation is that the obligation passing code is bundled together with the linked-list maintenance code (all of which is presented in assembler code). Because it is not clear what aspects of the linked-list code the obligation passing code depends on, it is difficult to apply this technique to other situations.

DISCLOSURE OF INVENTION

It has been discovered that solutions to a value recycling problem that we define herein facilitate implementations of computer programs that may execute as multithreaded computations in multiprocessor computers, as well as implementations of related shared data structures. Some exploitations of the techniques described herein allow non-blocking, shared data structures to be implemented using standard dynamic allocation mechanisms (such as `malloc` and `free`). Indeed, we present several exemplary realizations of dynamic-sized, non-blocking shared data structures that are not prevented from future memory reclamation by thread failures and which depend (in some implementations) only on widely-available hardware support for synchronization. Some exploitations of the techniques described herein allow *non-blocking*, indeed even *lock-free* or *wait-free*, implementations of dynamic storage allocation for shared data structures. Shared data structures that may benefit from the described techniques may themselves exhibit non-blocking, lock-free or wait-free properties, though need not in all implementations or modes of use. In some exploitations, our work provides a way to manage dynamically allocated memory in a non-blocking manner without depending on garbage collection. For example, techniques described herein may be exploited to manage dynamically allocated memory in a non-blocking manner in or for the implementation of a garbage collector itself.

A variety of solutions to the proposed value recycling problem may be implemented. A class of general solutions to value recycling is described in the context of an illustration we call the Repeat Offender Problem (ROP), including illustrative Application Program Interfaces (APIs) defined in terms of the ROP terminology. Furthermore, specific solutions, implementations and algorithm, including a Pass-The-Buck (PTB) implementation are described.

Solutions to the proposed value recycling problem can be applied in a variety of ways to implement dynamic-sized data structures. For example, specific solutions are illustrated in the context of particular shared data structures and algorithms, e.g., a lock-free FIFO queue for which we demonstrate true dynamic sizing. In

some cases, data structure implementations may be directly coded in ways that exploit the value recycling techniques described herein. In others, a *single-word lock-free reference counting* (SLFRC) technique (which builds on value recycling solutions) may be employed to transform, in a straight-forward manner, many lock-free data structure implementations that assume garbage collection (i.e., which do not explicitly free memory) into dynamic-sized data structures.

In some embodiments in accordance with the present invention, a method of supporting dynamic-sizing in a lock-free implementation of a shared data structure is provided. The method includes, in a thread of a multithreaded computation, dereferencing a pointer read from the shared data structure after successfully posting a guard on a value that encodes the pointer and freeing storage referenced by the value only after storage referenced by the pointer is not in the shared data structure and no successfully posted guard thereon remains uncancelled. In general, storage qualified for deallocation may be freed or reused, depending on the exploitation. For example, in some variations, storage referenced by a deallocation qualified value may be returned to a storage pool for reuse and freed as desired. In some variations, storage referenced by a deallocation qualified value may be directly freed to an operating system or memory allocator. Synchronization constructs such as a Compare-and-Swap (CAS) operation, a Load-Linked/Store-Conditional (LL/SC) operation pair and/or transactional sequences (e.g., as mediated by hardware transactional memory) may be employed in various realizations.

BRIEF DESCRIPTION OF DRAWINGS

The present invention may be better understood, and its numerous objects, features, and advantages made apparent to those skilled in the art by referencing the accompanying drawings.

FIG. 1 is a depicts a shared memory multiprocessor configuration that serves as a useful illustrative environment for describing operation of some shared object implementations in accordance with the present invention.

FIG. 2 illustrates transitions for a value *v* in accordance with one Repeat Offender Problem (ROP) formulation of value recycling.

FIG. 3 presents an I/O automaton specifying an exemplary formulation of the Repeat Offender Problem (ROP). The I/O automaton specifies environment and ROP output actions as well as state variables and transitions, including pre-conditions and post-conditions for various actions.

FIG. 4 is a timing diagram that illustrates interesting cases for an exemplary Pass The Buck (PTB) implementation of value recycling.

The use of the same reference symbols in different drawings indicates similar or identical items.

MODE(S) FOR CARRYING OUT THE INVENTION

A versatile mechanism has been developed for managing values shared amongst threads of a multithreaded computation. In some important exploitations, certain values so managed encode pointers to storage that is dynamically allocated, reused and/or freed in a computational system. Accordingly, techniques of the present invention provide a useful framework for supporting memory management in dynamic-sized data structures (i.e., those that can grow and shrink over time). Because some implementations of these techniques exhibit strong non-blocking properties (including, in some cases, *wait-free* properties), the techniques are particularly attractive for use in connection with non-blocking implementations of dynamic-sized, data structures. Indeed, a variety of applications to *lock-free* data structures are described herein.

However, while persons of ordinary skill in the art will recognize that the described techniques may be exploited in connection with data structures and/or algorithms that are non-blocking, indeed even lock-free or wait-free, based on the description herein, persons of ordinary skill in the art will also recognize that the described techniques may be exploited in connection with data structures and/or algorithms that are not necessarily non-blocking or for which not all modes of operation or use are non-blocking. Accordingly, descriptions herein made in the context of lock-free data structures are merely illustrative and provide a useful descriptive context in which the broader significance of the inventive techniques may be better appreciated.

As a result, descriptions of lock-free data structure exploitations should not be taken as limiting. Indeed, descriptions of exploitations in which managed values encode pointers should not be taken as limiting. As before, techniques for management of values that encode pointers simply provide useful descriptive context in which the broader significance of the inventive techniques may be better appreciated. Persons of ordinary skill in the art will appreciate, based on the description herein, that the inventive techniques are more generally applicable. In some exploitations, values so managed may include non-pointer values. For example, techniques of the present invention may be employed in the avoidance of ABA hazards. In some cases, avoided ABA hazards may involve non-pointer values. In others, avoided ABA hazards may involve pointer values and/or lock-free data structures. For example, one exploitation described herein illustrates use of the inventive techniques for avoidance of ABA hazards without version numbering commonly employed in the art.

Therefore, in view of the above, and without limitation, certain illustrative exploitations of the inventive techniques are described with particular attention to dynamic-sizing of lock-free data structures. Such illustrative exploitations should be viewed only as useful descriptive context, as the invention is defined solely by the claims that follow.

Dynamic-Sized Lock-Free Data Structures

In general, lock-free data structures avoid many of the problems associated with the use of locking, including convoying, susceptibility to failures and delays, and, in real-time systems, priority inversion. A *lock-free* implementation of a data structure provides that after a finite number of steps of any operation on the data structure, *some* operation completes. For reference, a *wait-free* implementation of a data structure provides

that after a finite number of steps of any operation on the data structure, *that* operation completes. Both lock-free implementations and wait-free implementations fall within the broader class of non-blocking implementations, though *wait-freedom* is clearly the stronger non-blocking property. Both of the preceding definitions (i.e., *lock-freedom* and *wait-freedom*) tend to preclude the use of locks to protect the data structure, because a thread can take an unbounded number of steps without completing an operation if some other thread is delayed or fails while holding a lock the first thread requires.

Lock-free data structures present design challenges that are well recognized in the art and that highlight advantages of the inventive techniques, although such techniques are more broadly applicable to data structures and/or algorithms that are not necessarily non-blocking or which may exhibit stronger or weaker non-blocking properties. Therefore, without loss of generality, we focus illustratively on lock-free data structures.

In general, the difficulty of designing lock-free data structures is reflected in numerous papers in the literature describing clever and subtle algorithms for implementing relatively mundane data structures such as stacks, queues, and linked lists. There are a variety of reasons that dynamic-sized data structures are challenging to implement in a lock-free manner. For example, before freeing an object that is part of a dynamic-sized data structure (e.g., a node of a linked list), it is important to ensure that no thread will subsequently modify the object. Otherwise, a thread might corrupt an object allocated later that happens to reuse some of the memory used by the first object. Furthermore, in some systems, even read-only accesses of freed objects can be problematic: the operating system may remove the page containing the object from the thread's address space, causing the subsequent access to crash the program because the address is no longer valid. In general, the use of locks makes it relatively easy to ensure that freed objects are not subsequently accessed, because we can prevent access by other threads to the data structure (or parts thereof) while removing objects from it. In contrast, without locks, multiple operations may access the data structure concurrently, and a thread cannot determine whether other threads are already committed to accessing the object that it wishes to free. This is the root of one problem that our work aims to address.

Our techniques build on a problem specification that we call the *Repeat Offender Problem* (ROP). In some variations, the problem may be specified more generally in terms of value recycling. Solutions to the problem can be used to design dynamic-sized, lock-free data structures that can free memory to the operating system without placing special restrictions on the memory allocation mechanism. Most previous dynamic-sized lock-free data structure implementations do not allow memory resources used by the data structure to be reclaimed and reused for other purposes. ROP solutions are useful in achieving truly dynamic-sized lock-free data structures that can continue to reclaim memory even in the face of thread failures. Our solution is implementable in most modern shared memory multiprocessors.

FIG. 1 depicts a shared memory multiprocessor configuration in which techniques of the present invention may be employed. In particular, **FIG. 1** depicts a pair of processors **111** and **112** that access storage **140**. Storage **140** includes a shared storage portion **130** and local storage portions **121** and **122**, respectively accessible by execution threads executing on processors **111** and **112**. In general, the multiprocessor

configuration is illustrative of a wide variety of physical implementations, including implementations in which the illustrated shared and local storage portions correspond to one or more underlying physical structures (e.g., memory, register or other storage), which may be shared, distributed or partially shared and partially distributed.

Accordingly, the illustration of **FIG. 1** is meant to exemplify an architectural view of a multiprocessor configuration from the perspective of execution threads, rather than any particular physical implementation. Indeed, in some realizations, data structures encoded in shared storage portion **130** (or portions thereof) and local storage (e.g., portion **121** and/or **122**) may reside in or on the same physical structures. Similarly, shared storage portion **130** need not correspond to a single physical structure. Instead, shared storage portion **130** may correspond to a collection of sub-portions each associated with a processor, wherein the multiprocessor configuration provides communication mechanisms (e.g., message passing facilities, bus protocols, etc.) to architecturally present the collection of sub-portions as shared storage. Furthermore, local storage portions **121** and **122** may correspond to one or more underlying physical structures including addressable memory, register, stack or other storage that are architecturally presented as local to a corresponding processor. Persons of ordinary skill in the art will appreciate a wide variety of suitable physical implementations whereby an architectural abstraction of shared memory is provided. Realizations in accordance with the present invention may employ any such suitable physical implementation.

In view of the foregoing and without limitation on the range of underlying physical implementations of the shared memory abstraction, operations on a shared object may be better understood as follows. Memory location **131** contains a pointer A that references an object **132** in shared memory. One or more pointers such as pointer A is (are) employed in a typical multithreaded computation. Local storage **134** encodes a pointer p_1 that references object **132** in shared memory. Local storage **135** also encodes a pointer p_2 that references object **132** in shared memory. In this regard, **FIG. 1** illustrates a state, $*p_1 == *A \;\&\&\; *p_2 == *A$, consistent with successful completion of load-type operations that bring a copies of pointer value A into local storage of two threads of a multithreaded computation.

As a general matter, **FIG. 1** sets up a situation in which operations of either thread may use (e.g. deference) their respective locally encoded value (e.g., pointer p_1 or p_2). Unfortunately, either thread may take action that recycles the value (e.g., qualifying, or freeing, object 132 for reuse) without coordination with the other thread. Absent coordination, operation of the multithreaded computation may be adversely affected. To see why this is so, and to explain our solution(s), we now turn to a simple data structure example.

Simple Example: A Lock-Free Stack

To illustrate the need for and use of techniques described herein, we consider a simple example: a lock-free integer stack implemented using the compare-and-swap (CAS) instruction. We first present a somewhat naive implementation, and then explain two problems with that implementation. We then show how to address these problems using value recycling techniques described herein. The following preliminaries apply to each of the algorithms presented.

Preliminaries

Our algorithms are presented in a C/C++-like pseudocode style, and should be self-explanatory. For convenience, we assume a shared-memory multiprocessor with sequentially consistent memory. We further assume that the multiprocessor supports a compare-and-swap (CAS) instruction that accepts three parameters: an *address*, an *expected* value, and a *new* value. The CAS instruction atomically compares the contents of the address to the expected value, and, if they are equal, stores the new value at the address and returns TRUE. If the comparison fails, no changes are made to memory, and the CAS instruction returns FALSE.

Suitable modifications for other programming styles, other memory consistency modes, other multiprocessor architectures and other synchronization facilities provided by other instruction sets and/or memory architectures/interfaces, are straightforward. Based on the specific examples presented herein, persons of ordinary skill in the art will appreciate a variety of such suitable modifications.

A Somewhat Naive Implementation and Its Pitfalls

A straightforward implementation approach for our lock-free integer stack is to represent the stack as a linked list of nodes, with a shared pointer—call it TOS—that points to the node at the top of the stack. In this approach, pushing a new value involves allocating a new node, initializing it with the value to be pushed and the current top of stack, and using CAS to atomically change TOS to point to the new node (retrying if the CAS fails due to concurrent operations succeeding). Popping is similarly simple: we use CAS to atomically change TOS to point to the second node in the list (again retrying if the CAS fails), and retrieve the popped value from the removed node. Unless we have GC to reclaim the removed node, we must explicitly free it to avoid a memory leak. Code for this (incorrect) approach follows:

```
struct nodeT {int val; nodeT *next;}
shared variable nodeT *TOS initially NULL;

Push(int v) {
1:  nodeT *oldtos, *newnode = malloc(sizeof(nodeT));
2:  newnode->val = v;
3:  do {
4:    oldtos = *TOS;
5:    newnode->next = oldtos;
6:  } while (!CAS(TOS,oldtos,newnode));
}

int Pop() {
7:   nodeT *oldtos, *newtos;
8:   do {
9:     oldtos = *TOS;
10:    if (oldtos == NULL) return "empty";
11:    newtos = oldtos->next;
12:  } while (!CAS(TOS,oldtos,newtos));
13:  int val = oldtos->val;
14:  free(oldtos);
15:  return val;
}
```

The first problem with the preceding stack implementation is that it allows a thread to access a freed node. To see why, observe that a thread *p* executing the Pop code at line 11 accesses the node it previously observed (at line 9) to be at the top of the stack. However, if another thread *q* executes the entire Pop operation between the times *p* executes lines 9 and 11, then it will free that node (line 14) and *p* will access a freed node.

The second problem is more subtle. This problem is widely known as the ABA problem, because it involves a variable changing from one value (A) to another (B), and subsequently back to the original value (A). The problem is that CAS does not distinguish between this situation and the one in which the variable does not change at all. The ABA problem manifests itself in the preceding stack implementation as follows. Suppose the stack currently contains nodes 1 and 2 (with node 1 being at the top of the stack). Further suppose that thread *p*, executing a Pop operation reads a pointer to node 1 from TOS at line 9, and then reads a pointer from node 1 to node 2 at line 11, and prepares to use CAS to atomically change TOS from pointing to node 1 to pointing to node 2. Now, let us suspend thread *p* for a moment. In the meantime, thread *q* executes the following sequence of operations: First *q* executes an entire Pop operation, removing and freeing node 1. Next, *q* executes another Pop operation, and similarly removes and deletes node 2. Now the stack is empty. Next *q* pushes a new value onto the stack, allocating node 3 for this purpose. Finally, *q* pushes yet another value onto the stack, and in this last operation, happens to allocate node 1 again (observe that node 1 was previously freed, so this is possible). Now, TOS points to node 1, which points to node 3. At this point, *p* resumes execution and executes its CAS, which succeeds in changing TOS from pointing to node 1 to pointing to node 2. This is incorrect, as node 2 has been freed (and may have subsequently been reallocated and reused for a different purpose). Further, note that node 3 has been lost from the stack. The root of the problem is that *p*'s CAS did not detect that TOS had changed from pointing to node 1 and later changed so that it was again pointing to node 1. This is the dreaded ABA problem. Although values A and B encode pointers in the preceding example, the ABA problem may affect non-pointer values as well.

Our Mechanisms: PostGuard and Liberate

We provide mechanisms that allow us to efficiently overcome both of the problems described above without relying on GC. Proper use of these mechanisms allows programmers to prevent memory from being freed while it might be accessed by some thread. In this subsection, we describe how these mechanisms should be used, and illustrate such use for the stack example.

The basic idea is that before dereferencing a pointer, a thread *guards* the pointer, and before freeing memory, a thread checks whether a pointer to the memory is guarded. For these purposes, we provide two functions, `PostGuard(ptr)` and `Liberate(ptr)`. `PostGuard` takes as an argument a pointer to be guarded. `Liberate` takes as an argument a pointer to be checked and returns a (possibly empty) set of pointers that it has determined are safe to free (at which time these pointers are said to be *liberated*). Thus, whenever a thread wants to free memory, instead of immediately invoking free, it passes the pointer to `Liberate`, and then invokes free on each pointer in the set returned by `Liberate`.

The most challenging aspect of using our mechanisms is that simply guarding a pointer is not sufficient to ensure that it is safe to dereference that pointer. The reason is that another thread might liberate and free a pointer after some thread has decided to guard that pointer, but before it actually does so. As explained in more detail below, `Liberate` never returns a pointer that is not safe to free, provided the programmer guarantees the following property:

> At the moment that a pointer is passed to `Liberate`, any thread that might dereference this instance of the pointer has already guarded it, and will keep the pointer guarded until after any such dereferencing.

Note that a particular pointer can be repeatedly allocated and freed, resulting in multiple *instances* of that pointer. Thus, this property refers to threads that might dereference a pointer before that same pointer is subsequently allocated again.

If a thread posts a guard on a pointer, and subsequently determines that the pointer has not been passed to `Liberate` since it was last allocated, then we say that the guard *traps* the pointer until the guard is subsequently posted on another pointer, or removed from this pointer. We have found this terminology useful in talking about algorithms that use our mechanisms. It is easy to see that the programmer can provide the guarantee stated above by ensuring that the algorithm never dereferences a pointer that is not trapped.

We have found that the following simple and intuitive pattern is often useful for achieving the required guarantee. First, a thread passes a pointer to `Liberate` only after it has determined that the memory block to which it points is no longer in the shared data structure. Given this, whenever a thread reads a pointer from the data structure in order to dereference it, it posts a guard on that pointer, and then attempts to determine that the memory block is still in the data structure. If it is, then the pointer has not yet been passed to `Liberate` and so it is safe to dereference the pointer; if not the thread retries. Determining whether a block is still in the data structure is sometimes as simple as rereading the pointer (for example, in the stack example presented next, we reread `TOS` to ensure that the pointer is the same as the one we guarded; see lines 9c and 9d in the exemplary code below.)

Using Our Mechanisms to Fix the Naive Stack Algorithm

In the exemplary code that follows, we present stack code modified to make the required guarantee.

```
struct nodeT {int val; nodeT *next;}
shared variable nodeT *TOS initially NULL;

Push(int v) {
1:  nodeT *oldtos, *newnode = malloc(sizeof(nodeT));
2:  newnode->val = v;
3:  do {
4:    oldtos = *TOS;
5:    newnode->next = oldtos;
6:  } while (!CAS(TOS,oldtos,newnode));
}
```

```
int Pop() {
7:   nodeT *oldtos, *newtos;
8:   do {
9a:    do {
9b:      oldtos = *TOS;
9c:      PostGuard(oldtos);
9d:    } while (*TOS != oldtos);
10:    if (oldtos == NULL) return "empty";
11:    newtos = oldtos->next;
12:  } while (!CAS(TOS,oldtos,newtos));
13:  int val = oldtos->val;
14a: PostGuard(NULL);
14b: for (nodeT *n in Liberate(oldtos))
14c:   free(n);
15:  return val;
}
```

To see how the modified code makes this guarantee, suppose that a thread *p* passes a pointer to node 1 to `Liberate` (line 14b) at time *t*. Prior to *t*, *p* changed `TOS` to a node other than node 1 or to `NULL` (line 12), and thereafter, until node 1 is liberated, freed and reallocated, `TOS` does not point to node 1. Suppose that after time *t*, another thread *q* dereferences (at line 11 or line 13) that instance of a pointer to node 1. When *q* last executes line 9d, at time *t'*, prior to dereferencing the pointer to node 1, *q* sees `TOS` pointing to node 1. Therefore, *t'* must have preceded *t*. Prior to *t'*, *q* guarded its pointer to node 1 (line 9c), and keeps guarding that pointer until after it dereferences it, as required. Note that the pointer is guarded until *q* executes line 14a (which stands down the guard) or line 9c (which effectively reassigns the guard to another post).

Guarantees of `PostGuard` and `Liberate`

While the above descriptions are sufficient to allow a programmer to correctly apply our mechanisms to achieve dynamic-sized data structures, it may be useful to understand in more detail the guarantees that are provided by the `Liberate` function. Below we describe those guarantees, and argue that they are sufficient, when they are used properly as described above, to prevent freed pointers from being dereferenced.

We say that a pointer *begins escaping* when `Liberate` is invoked with that pointer. Every liberated pointer—that is, every pointer in the set returned by a `Liberate` invocation—is guaranteed to have the following properties:

- It previously began escaping.
- It has not been liberated (by any `Liberate` invocation) since it most recently began escaping.
- It has not been guarded continuously by any thread since it most recently began escaping.

If pointers are only freed after they are returned by `Liberate`, the first two conditions guarantee that every instance of a pointer is freed at most once. They are sufficient for this purpose because threads only pass pointers to `Liberate` when they would have, in the straightforward (but defective) code, freed the pointers, and threads free only those pointers returned by `Liberate` invocations.

The last condition guarantees that a pointer is not liberated while it might still be dereferenced. To see that this last condition is sufficient, recall that the programmer must guarantee that any pointer passed to

`Liberate` at time *t* will be dereferenced only by threads that already guarded the pointer at time *t* and will keep the pointer guarded continuously until after such dereferencing. The last condition prevents the pointer from being liberated while any such thread exists.

Representative Application Programming Interface (API)

In this section, we present an application programming interface (API) for the guarding and liberating mechanisms illustrated in the previous section. This API is more general than the one used in the previous section. In particular, it allows threads to guard multiple pointers simultaneously.

Our API uses an explicit notion of *guards*, which are *posted* on pointers. In this API, a thread invokes `PostGuard` with both the pointer to be guarded and the guard to post on the pointer. We represent a guard by an `int`. A thread can guard multiple pointers by posting different guards on each pointer. A thread may *hire* or *fire* guards dynamically, according to the number of pointers it needs to guard simultaneously, using the `HireGuard` and `FireGuard` functions. We generalize `Liberate` to take a *set* of pointers as its argument, so that many pointers can passed to `Liberate` in a single invocation. The signatures of all these functions are shown below.

```
typedef guard int;
typedef ptr_t (void *);
void PostGuard(guard g, ptr_t p);
guard HireGuard();
void FireGuard(guard g);
set[ptr_t] Liberate(set[ptr_t] S);
```

[1001] Below we examine each function in more detail.

Detailed Function Descriptions

`void PostGuard(guard g, ptr_t p)`

Purpose:	Posts a guard on a pointer.
Parameters:	The guard `g` and the pointer `p`; `g` must have been hired and not subsequently fired by the thread invoking this function.
Return value:	None.
Remarks:	If `p` is `NULL` then `g` is not posted on any pointer after this function returns. If `p` is not `NULL`, then `g` is posted on `p` from the time this function returns until the next invocation of `PostGuard` with the guard `g`.

`guard HireGuard()`

Purpose:	"Acquire" a new guard.
Parameters:	None.
Return value:	A guard.
Remarks:	The guard returned is *hired* when it is returned. When a guard is hired, either it has not been hired before, or it has been fired since it was last hired.

`void FireGuard(guard g)`

Purpose:	"Release" a guard.

Parameters:	The guard `g` to be fired; `g` must have been hired and not subsequently fired by the thread invoking this function.
Return value:	None.
Remarks:	`g` is fired when `FireGuard(g)` is invoked.

`set[ptr_t] Liberate(set[ptr_t] S)`

Purpose:	Prepare pointers to be freed.
Parameters:	A set of pointers to liberate. Every pointer in this set must either never have begun escaping or must have been liberated since it most recently began escaping. That is, no pointer was in any set passed to a previous `Liberate` invocation since it was most recently in the set returned by some `Liberate` operation.
Return value:	A set of liberated pointers.
Remarks:	The pointers in the set `S` *begin escaping* when `Liberate(S)` is invoked. The pointers in the set returned are *liberated* when the function returns. Each liberated pointer must have been contained in the set passed to some invocation of `Liberate`, and not in the set returned by any `Liberate` operation after that invocation. Furthermore, `Liberate` guarantees for each pointer that it returns that no guard has been posted continuously on the pointer since it was most recently passed to some `Liberate` operation.

Comments on API design

We could have rolled the functionality of hiring and firing guards into the `PostGuard` operation. Instead, we kept this functionality separate to allow implementations to make `PostGuard`, the most common operation, as efficient as possible. This separation allows the implementation more flexibility in managing resources associated with guards because the cost of hiring and firing guards can be amortized over many `PostGuard` operations.

In some applications, it may be desirable to be able to quickly "mark" a value for liberation, without doing any of the work of liberating the value. Consider, for example, an interactive system in which user threads should not execute relatively high-overhead "administrative" work such as liberating values, but additional processor(s) may be available to perform such work. In such a case, it may be desirable to provide two (or more) versions of `Liberate`, where a *quick* version simply hands off all the values it is passed and returns the empty set.

Finally, our terminology is somewhat arbitrary here. In general, the relevant concepts may be expressed differently (or more generally) in other implementations. For example, rather than posting a guard on a value, a given implementation may be described in terms of an `Announce` operation that announces an intention to use the value. Similarly, functionality corresponding to that of guards may be provided using "handles" or other facilities for making such announcements. Liberating constructs may be implemented or represented in terms of a "cleaner" operation and related states. In short, many variations may be made without departing from central aspects of our inventive techniques.

The Repeat Offender Problem

We now more formally define a *Repeat Offender Problem* (ROP), which captures some important properties of the support mechanisms for nonblocking memory management described herein. ROP is defined with

respect to a set of *values*, a set of *application clients*, and a set of *guards*. Each value may be `free`, `injail`, or `escaping`; initially, all values are `free`. An application-dependent external `Arrest` action can cause a `free` value to become `injail` at any time. A client can help `injail` values to begin escaping, which causes them to become `escaping`. Values that are `escaping` can finish escaping and become `free` again.

Clients can use values, but must never use a value that is `free`. A client can attempt to prevent a value v from escaping while it is being used by "posting a guard" on v. However, if the guard is posted too late, it may fail to prevent v from escaping. Thus, to safely use v, a client must ensure that v is `injail` at some time *after* it posted a guard on v. Clients can hire and fire guards dynamically, according to their need.

ROP solutions can be used by threads (clients) to avoid dereferencing (using) a pointer (value) to an object that has been freed. In this context, an `injail` pointer is one that has been allocated (arrested) since it was last freed, and can therefore be used.

ROP solutions provide the following procedures: A client *hires* a guard by invoking `HireGuard()`, and it *fires* a guard g that it employs by invoking `FireGuard(g)`. A ROP solution ensures that a guard is never simultaneously employed by multiple clients. A client *posts* a guard g on a value v by invoking `PostGuard(g,v)`; this removes the guard from any value it previously guarded. In the implementation illustrated, a special `null` value is used to remove the guard from the previously guarded value without posting the guard on a new value. A client may not post (or remove) a guard that it does not currently employ. A client helps a set of values S to *begin escaping* by invoking `Liberate(S)`; the application must ensure that each value in S is `injail` before this call, and the call causes each value to become `escaping`. The `Liberate` procedure returns a (possibly different) set of `escaping` values causing them to be *liberated*; each of these values becomes `free` on the return of this procedure. These transitions are summarized in **FIG. 2**. A ROP solution does not implement the functionality of the `Arrest` action—this is application-specific, but the ROP specification models arrests in order to know when a `free` value becomes `injail`.

If a guard g is posted on a value v, and v is `injail` at some time t after g is posted on v and before g is subsequently removed or reposted on a different value, then we say that g *traps* v from time t until g is removed or reposted. Of course, subsequent removal or reposting is not a requirement for *trapping*. Accordingly, if g is never removed or reposted g traps v at a time t and all later times. The operational specification of the main correctness condition for ROP is that it does not allow a value to escape (i.e., become `free`) while it is trapped.

A precise formulation of ROP is given by the I/O automaton shown in **FIG. 3**, explained below. Of course, any of a variety of implementations in accordance with the I/O automaton are suitable. We begin by adopting some notational conventions.

Notational Conventions: Unless otherwise specified, p and q denote clients (threads) from P, the set of all clients (threads); g denotes a guard from G, the set of all guards; v denotes a value from V, the set of all values,

and S and T denote sets of values (i.e., subsets of V). We assume that V contains a special `null` value that is never used, arrested, or passed to liberate.

The automaton consists of a set of environment actions and a set of ROP output actions. Each action consists of a *precondition* for performing the action and the *effect* on state variables of performing the action. Most environment actions are invocations of ROP operations, and are paired with corresponding ROP output actions that represent the system's response to the invocations. In particular, the $\texttt{PostInv}_p\texttt{(g,v)}$ action models client p invoking `PostGuard(g,v)`, and the $\texttt{PostResp}_p\texttt{()}$ action models the completion of this procedure. The $\texttt{HireInv}_p\texttt{()}$ action models client p invoking `HireGuard()`, and the corresponding $\texttt{HireResp}_p\texttt{(g)}$ action models the system assigning guard g to p. The $\texttt{FireInv}_p\texttt{(g)}$ action models client p calling `FireGuard(g)`, and the $\texttt{FireResp}_p\texttt{()}$ action models the completion of this procedure. The $\texttt{LiberateInv}_p\texttt{(S)}$ action models client p calling `Liberate(S)` to help the values in S start escaping, and the $\texttt{LiberateResp}_p\texttt{(T)}$ action models the completion of this procedure with a set of values T that have finished escaping. Finally, the `Arrest(v)` action models the environment arresting value v.

The state variable *status[v]* records the current status of value v, which can be `free`, `injail`, or `escaping`. Transitions between the status values are caused by calls to and returns from ROP procedures, as well as by the application-specific `Arrest` action, as described above. The *post* variable maps each guard to the value (if any) it currently guards. The pc_p variable models control flow of client p, for example ensuring that p does not invoke a procedure before the previous invocation completes; pc_p also encodes parameters passed to the corresponding procedures in some cases. The $guards_p$ variable represents the set of guards currently employed by client p. The *numescaping* variable is an auxiliary variable used to specify nontriviality properties, as discussed later. Finally, *trapping* maps each guard g to a boolean value that is true iff g has been posted on some value v, and has not subsequently been reposted (or removed), and at some point since the guard was posted on v, v has been `injail` (i.e., it captures the notion of guard g trapping the value on which it has been posted). This is used by the `LiberateResp` action to determine whether v can be returned. Recall that a value should not be returned if it is trapped.

Preconditions on the invocation actions specify assumptions about the circumstances under which the application invokes the corresponding ROP procedures. Most of these preconditions are mundane well-formedness conditions, such as the requirement that a client posts only guards that it currently employs. The precondition for `LiberateInv` captures the assumption that the application passes only `injail` values to `Liberate`, and the precondition for the `Arrest` action captures the assumption that only `free` values are arrested. In general, a determination of how these guarantees are made is a matter of design choice.

Preconditions on the response actions specify the circumstances under which the ROP procedures can return. Again, most of these preconditions are quite mundane and straightforward. The interesting case is the precondition of `LiberateResp`, which states that `Liberate` can return a value only if it has been passed to (some invocation of) `Liberate`, it has not subsequently been returned by (any invocation of) `Liberate`,

and no guard *g* has been continually guarding the value since the last time it was `injail`. Recall that this is captured by *trapping[g]*.

Desirable Properties

As specified so far, an ROP solution in which `Liberate` always returns the empty set, or simply does not terminate, is correct. Clearly, in the context motivating our work, such solutions are unacceptable because each `escaping` value represents a resource that will be reclaimed only when the value is liberated (returned by some invocation of `Liberate`). One might be tempted to specify that every value passed to a `Liberate` operation is eventually returned by some `Liberate` operation. However, without special operating system support, it is generally not possible to guarantee such a strong property in the face of failing threads. We do not specify here a particular nontriviality condition, as we do not want to unduly limit the range of solutions. Instead, we discuss some properties that might be useful in specifying nontriviality properties for proposed solutions.

The state variable *numescaping* counts the number of values that are currently escaping (i.e., that have been passed to some invocation of `Liberate` and have not subsequently been returned from any invocation of `Liberate`). If we require a solution to ensure that *numescaping* is bounded by some function of application-specific quantities, we exclude the trivial solution in which `Liberate` always returns the empty set. However, because this bound necessarily depends on the number of concurrent `Liberate` operations, and the number of values each `Liberate` operation is invoked with, it does not exclude the solution in which `Liberate` never returns.

A combination of a boundedness requirement and some form of progress requirement on `Liberate` operations seems to be the most appropriate way to specify the nontriviality requirement. Recall that we have defined a general value recycling problem in terms of Repeat Offender style terminology (e.g., posting guards, liberation and states such as *injail* and *escaping*) and that we expect a variety of implementations and algorithms to solve that general problem. One such implementation includes the Pass The Buck (PTB) algorithm (detailed below), which for simplicity of description is also presented in Repeat Offender style terminology. Turning to the PTB algorithm, we can establish that PTB provides a bound on *numescaping* that depends on the number of concurrent `Liberate` operations. Because the bound (necessarily) depends on the number of concurrent `Liberate` operations, if an unbounded number of threads fail while executing `Liberate`, then an unbounded number of values can be escaping. We emphasize, however, that our implementation does *not* allow failed threads to prevent values from being freed in the future. This property is an important advantage over Treiber's approach (referenced above).

Our Pass The Buck algorithm has two other desirable properties: First, the `Liberate` operation is wait-free (that is, it completes after a bounded number of steps, regardless of the timing behavior of other threads). Thus, we can calculate an upper bound on the amount of time `Liberate` will take to execute, which is useful in determining how to schedule `Liberate` work. Finally, our algorithm has a property we call *value*

progress. Roughly, this property guarantees that a value does not remain *escaping* forever provided `Liberate` is invoked "enough" times (unless a thread fails while the value is escaping).

Dynamic-Sized Lock-Free Queues

In this section, we present two dynamic-sized lock-free queue implementations based on a widely used lock-free queue algorithm previously described by Michael and Scott. *See* M. Michael & M. Scott, *Nonblocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors, Journal or Parallel and Distributed Computing*, 51(1):1-26, 1998. Note that Michael and Scott's algorithm and data structure implemented generally in accordance therewith provide us with useful context to describe our additional inventive concepts. Nothing herein should be taken as a suggestion that our techniques are derived from, linked with, or limited to the algorithm, data structures or any design choices embodied in or by Michael and Scott's work.

In Michael and Scott's algorithm (hereafter M&S), a queue is represented by a linked list, and nodes that have been dequeued are placed in a "freelist" implemented in the style of Treiber. In the description that follows, we refer to such freelists as "memory pools" in order to avoid confusion between "freeing" a node—by which we mean returning it to the memory allocator through the `free` library routine—and placing a node on a freelist. In this approach, rather than freeing nodes to the memory allocator when they are no longer required, we place them in a memory pool from which new nodes can be allocated later. An important disadvantage of this approach is that data structures implemented this way are not truly dynamic-sized: after they have grown large and subsequently shrunk, the memory pool contains many nodes that cannot be reused for other purposes, cannot be coalesced, etc.

Our two queue implementations achieve dynamic-sizing in different ways. Algorithm 1 eliminates the memory pool, invoking the standard `malloc` and `free` library routines to allocate and deallocate nodes of the queue. Algorithm 2 does use a memory pool, but unlike M&S, the nodes in the memory pool can be freed to the system. We present our algorithms in the context of a transformed version of the M&S algorithm (*see* below). This "generic code" invokes additional procedures that must be instantiated to achieve full implementations. We first provide exemplary instantiations consistent with the original M&S algorithm. Then, we provide instantiations for our new algorithms. In this way, we illustrate true dynamic-sizing in the context of a familiar lock-free data structure design that does not itself provide a true dynamic sizing capability.

Note that although the M&S design does allow nodes to be added and removed from the queue, such nodes are added from, and removed to, a memory pool. Since no mechanism is provided to remove nodes from the memory pool, the amount of storage allocated for use by the queue is monotonic, non-decreasing. Accordingly, it is not really correct to describe the M&S design as a dynamic-sized lock-free data structure. Our work achieves a true dynamic-sized lock-free queue.

Michael and Scott's Algorithm

[1002] In general, the M&S design will be understood in the context of the following slightly transformed version which plays the role of a "generic code" base for the modified versions that follow. The M&S design builds on a queue data structure that will be understood as follows:

```
struct pointer_t { node_t *ptr; int version; }
struct node_t { int value; pointer_t next; }
struct queue_t { pointer_t Head, Tail; }

queue_t *newQueue() {
  queue_t *Q = malloc(sizeof(queue_t));
  node_t *node = allocNode();
  node->next.ptr = null;
  Q->Head.ptr = Q->Tail.ptr = node;
  return Q;
}

bool Enqueue(queue_t *Q, int value) {
1  node_t *node = allocNode();
2  if (node == null)
3    return FALSE;
4  node->value = value;
5  node->next.ptr = null;
6  while (TRUE) {
7    pointer_t tail;
8    GuardedLoad(&Q->Tail, &tail, 0);
9    pointer_t next = tail.ptr->next;
10   if (tail == Q->Tail) {
11     if (next.ptr == null) {
12       if (CAS(&tail.ptr->next, next, <node,next.version+1>))
13         break;
14     } else
15       CAS(&Q->Tail, tail, <next.ptr,tail.version+1>)
     }
   }
16 CAS(&Q->Tail, tail, <node.tail,version+1>)
17 Unguard(0);
18 return TRUE;
}

bool Dequeue(queue_t *Q, int *pvalue) {
19  while (TRUE) {
20    pointer_t head;
21    GuardedLoad(&Q->Head, &head, 0);
22    pointer_t tail = Q->Tail;
23    pointer_t next;
24    GuardedLoad(&head.ptr->next, &next, 1);
25    if (head == Q->Head) {
26      if (head.ptr == tail.ptr) {
27        if (next.ptr == null) {
28          Unguard(0);
29          Unguard(1);
30          return FALSE;
          }
31        CAS(&Q->Tail, tail, <next.ptr,tail.version+1>)
32      } else {
33        *pvalue = next.ptr->value;
34        if (CAS(&Q->Head, head, <next.ptr,head.version+1>))
```

```
35          break;
          }
        }
      }
36  Unguard(0);
37  Unguard(1);
38  deallocNode(head.ptr);
39  return TRUE;
}
```

The preceding generic code invokes four additional procedures, shown in *italics*, which are not specified in the generic code. Three variations on the M&S design can be achieved using three different implementations of the additional procedure sets. For completeness, a first set results in an implementation that corresponds to the original M&S design. In short, we get the original M&S algorithm by instantiating the following procedures:

```
node_t *allocNode() {
1  if (memory pool is empty)
2    return malloc(sizeof(node_t));
3  else {
4    return node removed from memory pool;
   }
}

void deallocNode(node_t *n) {
5  add n to memory pool
}

void GuardedLoad(pointer_t *s, pointer_t *t, int h) {
6  *t = *s;
7  return;
}

void Unguard(int h) {
8  return;
}
```

The *allocNode* and *deallocNode* procedures use a memory pool. The *allocNode* procedure removes and returns a node from the memory pool if possible and calls `malloc` if the Pool is empty. The *deallocNode* procedure puts the node being deallocated into the memory pool. As stated above, nodes in the memory pool cannot be freed to the system. Michael and Scott do not specify how nodes are added to and removed from the memory pool. Because the M&S design does not use value recycling technique such as that provided by solutions to the ROP, it has no notion of "guards." As a result, *GuardedLoad* is an ordinary load and *Unguard* is a no-op.

We do not discuss M&S in detail. *See* M. Michael & M. Scott, *Nonblocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors, Journal or Parallel and Distributed Computing*, 51(1):1-26, 1998 for such details. Instead, we discuss below the aspects that are relevant for our purposes.

Although nodes in the M&S memory pool have been deallocated, they cannot be freed to the system because some thread may still intend to perform a CAS on the node. Various problems can arise from accesses to

memory that has been freed. Thus, although it is not discussed at length by the authors, M&S' use of the memory pool is necessary for correctness. Because `Enqueue` may reuse nodes from the memory pool, M&S uses version numbers to avoid the ABA problem, in which a CAS succeeds even though the pointer it accesses has changed because the node pointed to was deallocated and then subsequently allocated. The version numbers are stored with each pointer and are atomically incremented each time the pointer is modified. This causes such "late" CAS's to fail, but it does not prevent them from being attempted.

The queue is represented by two node pointers: the `Head`, from which nodes are dequeued, and the `Tail`, where nodes are enqueued. The `Head` and `Tail` pointers are never `null`; the use of a "dummy" node ensures that the list always contains at least one node. When a node is deallocated, no path exists from either the `Head` or the `Tail` to that node. Furthermore, such a path cannot subsequently be established before the node is allocated again in an Enqueue operation. Therefore, if such a path exists, then the node is in the queue. Also, once a node is in the queue and its `next` field has become non-`null`, its `next` field cannot become `null` again until the memory that contains the node is subsequently reallocated, implying that the node has been freed before that time. These properties provide a basis to establish correctness of our dynamic-sized variants of the generic M&S design.

Algorithm 1 – Direct Dynamic Sizing (No Memory Pool)

As mentioned earlier, Algorithm 1 eliminates the memory pool, and uses `malloc` and `free` directly for memory allocation. As discussed below, Algorithm 1 also eliminates the ABA problem, and thus, the need for version numbers. Significantly and unlike the original M&S design, this feature allows designs based on Algorithm 1 to be used on systems that support CAS only on pointer-sized values.

For ease of understanding we present variations for Algorithm 1 in the context of the above "generic code." Algorithm 1, with true dynamic sizing, is achieved by instantiating the following variations of the additional procedures for use by the generic code:

```
Node_t *allocNode() {
1  return malloc(sizeof(node_t));
}

void deallocNode(node_t *n) {
2  for each m ∈ Liberate({n})
3    free(m);
}

void GuardedLoad(pointer_t *s, pointer_t *t, int g) {
4  while (TRUE) {
5    *t = *s;
6    if (t->ptr == null)
7      return;
8    PostGuard(guards[p][g], t->ptr);
9    if (*t == *s)
10     return;
   }
}
```

```
void Unguard(int g) {
11  PostGuard(guards[p][g], null);
}
```

Persons of ordinary skill in the art will, of course, recognize that separation of Algorithm 1 into additional procedures and generic code is arbitrary and is employed purely for illustration. Corresponding, indeed even technically equivalent, implementations may be achieved without resorting to this artificial pedagogical separation.

As explained below, the preceding procedures employ value recycling techniques in accordance with a ROP formulation. We assume that before accessing the queue, each thread *p* has hired two guards and stored identifiers for these guards in `guards[p][0]` and `guards[p][1]`. The *allocNode* procedure simply invokes `malloc`. However, because some thread may have a pointer to a node being deallocated, *deallocNode* cannot simply invoke `free`. Instead, *deallocNode* passes the node being deallocated to `Liberate` and then frees any nodes returned by `Liberate`. The properties of ROP ensure that a node is never returned by an invocation of `Liberate` while some thread might still access that node.

The *GuardedLoad* procedure loads a value from the address specified by its first argument and stores the value loaded in the address specified by its second argument. The goal of this procedure is to ensure that the value loaded is guarded by the guard specified by the third argument *before* the value is loaded. This goal is accomplished by a lock-free loop that retries if the value loaded changes after the guard is posted (`null` values do not have to be guarded, as they will never be dereferenced). As explained below, *GuardedLoad* helps ensure that guards are posted soon enough to trap the pointers they guard, and therefore to prevent the pointers they guard from being freed prematurely. The *Unguard* procedure removes the specified guard.

Correctness argument for Algorithm 1

Operation of Algorithm 1 corresponds to that of M&S, *except for issues involving memory allocation and deallocation.* To see this, observe that *GuardedLoad* implements an ordinary load, and *Unguard* does not affect any variables of the underlying M&S algorithm. Therefore, we need only argue that no instruction accesses a freed node. Because nodes are freed only after being returned by `Liberate`, it suffices to argue for each access to a node, that, at the time of the access, a pointer to the node has been continuously guarded since some point at which the node was in the queue (that is, a node is accessed only if it is trapped). As discussed earlier, if there is a path from either `Head` or `Tail` to a node, then the node is in the queue. As shown below, we can exploit code already included in M&S, together with the specialization code in Figure 6, to detect the existence of such paths.

We first consider the access at line 9 of `Enqueue`. In this case, the pointer to the node being accessed was acquired from the call to *GuardedLoad* at line 8. Because the pointer is loaded directly from `Tail` in this case, the load in line 9 of the Algorithm 1 implementation of *GuardedLoad* serves to observe a path (of length one) from `Tail` to the accessed node. The argument is similarly straightforward for the access at line 12 of `Enqueue` and the access in *GuardedLoad* when invoked from line 24 (`Dequeue`).

The argument for the access at line 33 of `Dequeue` is not as simple. First, observe that the load at line 9 of *GuardedLoad* (in the call at line 24 of `Dequeue`) determines that there is a pointer from the node specified by `Head.ptr` to the node accessed at line 33. Then, the test at line 25 determines that there is a pointer from `Head` to the node specified by `Head.ptr`. If these two pointers existed simultaneously at some point between the guard being posted as a result of the call at line 24 and the access at line 33, then the required path existed. As argued above, the node pointed to by `Head.ptr` is guarded and was in the queue at some point since the guard was posted in the call to *GuardedLoad* at line 21, and this guard is not removed or reposted before the execution of line 33. Therefore, relying on the properties of ROP, this node cannot be freed and reallocated in this interval. Also, in the M&S algorithm, a node that is dequeued does not become reachable from `Head` again before it has been reallocated by an `Enqueue`. Therefore, the load at line 25 confirmed that `Head` contained the same value continuously since the execution of line 21. This in turn implies that the two pointers existed simultaneously at the point at which the load in *GuardedLoad* invoked from line 24 was executed. The last part of this argument can be made much more easily by observing that the version number (discussed next) of `Head` did not change. However, we later observe that the version numbers can be eliminated from Algorithm 1, so we do not want to rely on them in our argument. This concludes our argument that Algorithm 1 never accesses freed memory.

Next, we show that the version numbers for the node pointers are unnecessary in our Algorithm 1. Apart from the overhead involved with managing these version numbers, the requirement that they are updated atomically with pointers renders algorithms that use them inapplicable in systems that support CAS only on pointer-sized values. Accordingly, the ability to eliminate version numbers is an important achievement in and of itself. In addition, since version numbers were "necessary" to avoid an ABA hazard, another useful exploitation of the invented techniques is now apparent, namely ABA hazard avoidance.

Eliminating version numbers in Algorithm 1

By inspecting the code for Algorithm 1, we can see that the only effect of the version numbers is to make some comparisons fail that would otherwise have succeeded. These comparisons are always between a shared variable V and a value previously read from V. The comparisons would fail anyway if V's pointer component had changed, and would succeed in any case if V had not been modified since the V was read. Therefore, version numbers change the algorithm's behavior only in the case that a thread p reads value A from V at time t, V subsequently changes to some other value B, and later still, at time t', V changes back to a value that contains the same pointer component as A, and p compares V to A. With version numbers, the comparison would fail, and without them it would succeed. We begin by establishing that version numbers do not affect the outcome of comparisons other than the one in line 9 of *GuardedLoad*. We deal with that case later.

We first consider cases in which A's pointer component is non-`null`. It can be shown for each shared pointer variable V in the algorithm that the node pointed to by A is freed and subsequently reallocated between times t and t' in this case. Furthermore, it can be shown that each of the comparisons mentioned above occurs only if a guard was posted on A before time t and is still posted when the subsequent comparison is performed, and

that the value read from *A* was in the queue at some point since the guard was posted when the comparison is performed. Because ROP prohibits nodes from being returned by `Liberate` (and therefore from being freed) in this case, this implies that these comparisons never occur in Algorithm 1.

We next consider the case in which *A*'s pointer component is `null`. The only comparison of a shared variable to a value with a `null` pointer is the comparison performed at line 12 of the `Enqueue` operation (because the `Head` and `Tail` never contain `null` and therefore neither do the values read from them). As argued earlier, the access at line 12 is performed only when the node being accessed is trapped. Also, as discussed earlier, the `next` field of a node in the queue does not become `null` again until the node is initialized by the next `Enqueue` operation to allocate that node. However, ROP ensures that the node is not returned from `Liberate`, and is therefore not subsequently freed and reallocated, before the guard is removed or reposted.

It remains to consider the comparison in line 9 of *GuardedLoad*, which can have a different outcome if version numbers are used than it would if they were not used. However, this does not affect the externally-observable behavior of the *GuardedLoad* procedure, and therefore does not affect correctness. The only property of the *GuardedLoad* procedure on which we have depended for our correctness argument is the following: *GuardedLoad* stores a value *v* in the location pointed to by its second argument such that *v* was in the location pointed to by *GuardedLoad*'s first argument at some point during the execution of *GuardedLoad* and that a guard was posted on (the pointer component of) *v* before that time and has not subsequently been reposted or removed. It is easy to see that this property is guaranteed by the *GuardedLoad* procedure, with or without version numbers.

Algorithm 2 – with Memory Pool

One drawback of Algorithm 1 is that every `Enqueue` and `Dequeue` operation involves a call to the `malloc` or `free` library routine (or other similar facility) introducing significant overhead. In addition, every `Dequeue` operation invokes `Liberate`, which is also likely to be expensive. Algorithm 2 overcomes these disadvantages by reintroducing the memory pool. However, unlike the M&S algorithm, nodes in the memory pool of Algorithm 2 can be freed to the system.

Algorithm 2 is achieved by instantiating the generic code (described above) with the same *GuardedLoad* and *Unguard* procedures used for Algorithm 1, though with modified *allocNode* and *deallocNode* procedures such as illustrated below:

```
Pointer_t Pool;

Node_t *allocNode() {
1   pointer_t oldPool, newPool;
2   while (TRUE) {
3     GuardedLoad(&Pool, &oldPool, 0);
4     if (oldPool.ptr == null) {
5       Unguard(0);
6       return malloc(sizeof(node_t));
      }
7     newPool = oldPool.ptr->next;
```

```
8     Unguard(0);
9     newPool.version = oldPool.version + 1;
10    if (CAS(&Pool, oldPool, newPool)) {
11      return oldPool.ptr;
      }
    }
}

void deallocNode(node_t *n) {
12 pointer_t oldPool, newPool;
13 while (TRUE) {
14   oldPool = Pool;
15   n->next.ptr = oldPool.ptr;
16   newPool.ptr = n;
17   newPool.version = oldPool.version + 1;
18   if (CAS(&Pool, oldPool, newPool))
19     return;
    }
}
```

As in the original M&S algorithm, the *allocNode* and *deallocNode* procedures, respectively, remove nodes from and add nodes to the memory pool. Unlike the original algorithm, however, the memory pool is implemented so that nodes can be freed. Thus, by augmenting Algorithm 2 with a policy that decides between freeing nodes and keeping them in the memory pool for subsequent use, a truly dynamic-sized implementation can be achieved.

The above procedures use a linked-list representation of a stack for a memory pool. This implementation extends Treiber's straightforward implementation by guarding nodes in the `Pool` before accessing them; this allows us to pass removed nodes to `Liberate` and to free them when returned from `Liberate` without the risk of a thread accessing a node after it has been freed. Our memory pool implementation is described in more detail below.

The node at the top of the stack is pointed to by a global variable `Pool`. We use the `next` field of each node to point to the next node in the stack. The *deallocNode* procedure uses a lock-free loop; each iteration uses CAS to attempt to add the node being deallocated onto the top of the stack. As in Treiber's implementation, a version number is incremented atomically with each modification of the `Pool` variable to avoid the ABA problem.

The *allocNode* procedure is more complicated. In order to remove a node from the top of the stack, *allocNode* determines the node that will become the new top of the stack. This is achieved by reading the `next` field of the node that is currently at the top of the stack. As before, we use a ROP-style value recycling solution to protect against the possibility of accessing (at line 7) a node that has been freed. Therefore, the node at the top of the stack is guarded and then confirmed by the *GuardedLoad* call at line 3. As in the easy cases discussed above for Algorithm 1, the confirmation of the pointer loaded by the call to *GuardedLoad* establishes that the pointer is trapped, because a node will not be passed to `Liberate` while it is still at the `Head` of the stack.

We have not specified when or how nodes are passed to `Liberate`. There are many possibilities and the appropriate choice depends on the application and system under consideration. Any of a variety of design

choices are suitable. One possibility is for the *deallocNode* procedure to liberate nodes when the size of the memory pool exceeds some fixed limit. Alternatively, we could have an independent "helper" thread that periodically (or routinely) checks the memory pool and decides whether to liberate some nodes in order to reduce the size of the memory pool. Such decisions could be based on the size of the memory pool or on other criteria. In general, there is no need for the helper thread to grow the memory pool because this will occur naturally. When there are no nodes in the memory pool, *allocNode* invokes `malloc` to allocate space for a new node.

The Pass The Buck Algorithm

In this section, we describe one value recycling solution, the *Pass The Buck* (PTB) algorithm. As before, the PTB algorithm is presented using ROP terminology. An important goal when designing PTB was to minimize the performance penalty to the application when no values are being liberated. That is, the `PostGuard` operation should be implemented as efficiently as possible, perhaps at the cost of a more expensive `Liberate` operation. Such solutions are desirable for at least two reasons. First, `PostGuard` is invoked by the application, so its performance impacts application performance. On the other hand, `Liberate` work can be done by a spare processor, or by a background thread, so that it does not directly impact application performance. Second, solutions that optimize `PostGuard` performance are desirable for scenarios in which values are liberated infrequently, but we must retain the ability to liberate them. An example is the implementation of a dynamic-sized data structure that uses a memory pool to avoid allocating and freeing objects under "normal" circumstances but can free elements of the memory pool when it grows too large. In this case, no liberating is necessary while the size of the data structure is relatively stable. With these goals in mind, we describe our Pass The Buck algorithm below.

```
struct {value val; int ver} HO_t
   // HO_t fits into CAS-able location
constant MG: max. number of guards
shared variable
   GUARDS: array[0..MG-1] of bool init false;
   MAXG: int init 0;
   POST: array[0..MG-1] of value init null;
   HNDOFF: array[0..MG-1] of HO_t init <null, 0>;

int HireGuard() {
1  int i = 0, max;
2  while (!CAS(&GUARDS[i],false,true))
3    i++;
4  while ((max = MAXG) < i)
5    CAS(&MAXG,max,i);
6  return i;
}

void FireGuard(int i) {
7  GUARDS[i] = false;
8  return;
}

void PostGuard(int i, value v) {
9  POST[i] = v;
```

```
10 return;
}

value set Liberate(value set vs) {
11 int i = 0;
12 while (i <= MAXG) {
13   int attempts = 0;
14   HO_t h = HNDOFF[i];
15   value v = POST[i];
16   if (v != null && vs->search(v)) {
17     while (true) {
18       if (CAS(&HNDOFF[i], h, <v, h.ver+1>)) {
19         vs->delete(v);
20         if (h.val != null) vs->insert(h.val);
21         break;
         }
22       attempts++;
23       if (attempts == 3) break;
24       h = HNDOFF[i];
25       if (attempts == 2 && h.val != null) break;
26       if (v != POST[i]) break;
       }
27   } else {
28     if (h.val != null && h.val != v)
29       if (CAS(&HNDOFF[i], h, <null, h.ver+1>))
30         vs->insert(h.val);
     }
31   i++;
   }
32 return vs;
}
```

Throughout the algorithm, the pointers to blocks of memory being managed are called *values*. The GUARDS array is used to allocate guards to threads. Here we assume a bound MAXG on the number of guards simultaneously employed. However, as later explained, we can remove this restriction. The POST array includes one location per guard, which holds the pointer value the guard is currently assigned to guard if one exists, and NULL otherwise. The HNDOFF array is used by `Liberate` to "hand off" responsibility for a value to another `Liberate` operation if the value has been trapped by a guard.

The `HireGuard` and `FireGuard` procedures essentially implement long-lived renaming. Specifically, for each guard g, we maintain an entry GUARDS [g], which is initially *false*. Thread p hires guard g by atomically changing GUARDS [g] from *false* (unemployed) to *true* (employed); p attempts this with each guard in turn until it succeeds (lines 2 and 3). The `FireGuard` procedure simply sets the guard back to *false* (line 7). The `HireGuard` procedure also maintains the shared variable MAXG, which is used by the `Liberate` procedure to determine how many guards to consider. `Liberate` considers every guard for which a `HireGuard` operation has completed. Therefore, it suffices to have each `HireGuard` operation ensure that MAXG is at least the index of the guard returned. This is achieved with the simple loop at lines 4 and 5.

`PostGuard` is implemented as a single store of the value to be guarded in the specified guard's POST entry (line 9), in accordance with our goal of making `PostGuard` as efficient as possible.

Some of the most interesting parts of the PTB algorithm lie in the `Liberate` procedure. Recall that `Liberate` should return a set of values that have been passed to `Liberate` and have not since been returned by `Liberate`, subject to the constraint that `Liberate` cannot return a value that has been continuously guarded by the same guard since before the value was most recently passed to `Liberate` (i.e., `Liberate` must not return trapped values).

`Liberate` is passed a set of values, and it adds values to and removes values from its value set as described below before returning (i.e., liberating) the remaining values in the set. Because we want the `Liberate` operation to be wait-free, if some guard g is guarding a value v in the value set of some thread p executing `Liberate`, then p must either determine that g is not trapping v or remove v from p's value set before returning that set. To avoid losing values, any value that p removes from its set must be stored somewhere so that, when the value is no longer trapped, another `Liberate` operation may pick it up and return it. The interesting details of PTB concern how threads determine that a value is not trapped, and how they store values while keeping space overhead for stored values low. Below, we explain the `Liberate` procedure in more detail, paying particular attention to these issues.

The loop at lines 12 through 31 iterates over all guards ever hired. For each guard g, if p cannot determine for some value v in its set that v is not trapped by g, then p attempts to "hand v off to g." If p succeeds in doing so (line 18), it removes v from its set (line 19) and proceeds to the next guard (lines 21 and 31). If p repeatedly attempts and fails to hand v off to g, then, as we explain below, v cannot be trapped by g, so p can move on to the next guard. Also, as explained in more detail below, p might simultaneously pick up a value previously handed off to g by another `Liberate` operation, in which case this value can be shown not to be trapped by g, so p adds this value to its set (line 20). When p has examined all guards (see line 12), it can safely return any values remaining in its set (line 32).

We describe the processing of each guard in more detail below. First, however, we present a central property of a correctness proof of this algorithm, which will aid the presentation that follows; this lemma is quite easy to see from the code and the high-level description given thus far.

Single Location Lemma: For each value v that has been passed to some invocation of `Liberate` and not subsequently returned by any invocation of `Liberate`, either v is handed off to exactly one guard, or v is in the value set of exactly one `Liberate` operation (but not both). Also, any value handed off to a guard or in the value set of any `Liberate` operation has been passed to `Liberate` and not subsequently returned by `Liberate`.

The processing of each guard g proceeds as follows: At lines 15 and 16, p determines whether the value currently guarded by g (if any)—call it v—is in its set. If so, p executes the loop at lines 17 through 26 in order to either determine that v is not trapped, or to remove v from its set. In order to avoid losing v in the latter case, p "hands v off to g" by storing v in `HNDOFF[g]`. In addition to the value, an entry in the `HNDOFF` array contains a version number, which, for reasons that will become clear later, is incremented with each modification of the entry. Because at most one value may be trapped by guard g at any time, a single location

HNDOFF [g] for each guard g is sufficient. To see why, observe that if p needs to hand v off because it is guarded, then the value (if any)—call it w—previously stored in HNDOFF [g] is no longer guarded, so p can pick w up and add it to its set. Because p attempts to hand off v only if v is in p's set, the Single Location Lemma implies that $v \neq w$. The explanation above gives the basic idea of our algorithm, but it is simplified. There are various subtle race conditions that must be avoided. Below, we explain in more detail how the algorithm deals with these race conditions.

To hand v off to g, p uses a CAS operation to attempt to replace the value previously stored in HNDOFF [g] with v (line 18); this ensures that, upon success, p knows which value it replaced, so it can add that value to its set (line 20). We explain later why it is safe to do so. If the CAS fails due to a concurrent `Liberate` operation, then p rereads HNDOFF [g] (line 24) and loops around to retry the handoff. There are various conditions under which we break out of this loop and move on to the next guard. Note in particular that the loop completes after at most three CAS attempts; see lines 13, 22, and 23. Thus our algorithm is wait-free. We explain later why it is safe to stop trying to hand v off in each of these cases.

We first consider the case in which p exits the loop due to a successful CAS at line 18. In this case, as described earlier, p removes v from its set (line 19), adds the previous value in HNDOFF [g] to its set (line 20), and moves on to the next guard (lines 21 and 31). An important part of understanding our algorithm is to understand why it is safe to take the previous value—call it w—of HNDOFF [g] to the next guard. The reason is that we read POST [g] (line 15 or 26) between reading HNDOFF [g] (line 14 or 24) and attempting the CAS at line 18. Because each modification to HNDOFF [g] increments its version number, it follows that w was in HNDOFF [g] when p read POST [g]. Also, recall that $w \neq v$ in this case. Therefore, when p read POST [g], w was not guarded by g. Furthermore, because w remained in HNDOFF [g] from that moment until the CAS, w cannot become trapped in this interval. This is because a value can become trapped only if it has not been passed to `Liberate` since it was last allocated, and all values in the HNDOFF array have been passed to some invocation of `Liberate` and not yet returned by any invocation of `Liberate`(and have therefore not been freed and reallocated since being passed to `Liberate`).

It remains to consider how p can break out of the loop without performing a successful CAS. In each case, p can infer that v is not trapped by g, so it can give up on its attempt to hand v off. If p breaks out of the loop at line 26, then v is not trapped by g at that moment simply because it is not even guarded by g. The other two cases (lines 23 and 25) occur only after a certain number of times around the loop, implying a certain number of failed CAS operations.

To see why we can infer that v is not trapped in each of these two cases, consider the timing diagram in **FIG. 4**. For the rest of this section, we use the notation v_p to indicate the value of thread p's local variable v in order to distinguish between the local variables of different threads. In **FIG. 4**, we construct an execution in which p fails its CAS three times. The bottom line represents thread p:

- at **(A)**, p reads HNDOFF [g] for the first time (line 14);
- at **(B)**, p's CAS fails;

- at (C), p rereads HNDOFF [g] at line 24; and
- so on for (D), (E), and (F).

Because p's CAS at (B) fails, some other thread q_0 executing `Liberate` performed a successful CAS after (A) and before (B). We choose one and call it (G). The arrows between (A) and (G) and between (G) and (B) indicate that we know (G) comes after (A) and before (B). Similarly, some thread q_1 executes a successful CAS on HNDOFF [g] after (C) and before (D)—call it (H); and some thread q_2 executes a successful CAS on HNDOFF [g] after (E) and before (F)—call it (I). Threads q_0 through q_2 might not be distinct, but there is no loss of generality in treating them as if they were.

Now, consider the CAS at (H). Because every successful CAS increments the version number field of HNDOFF [g], q_1's previous read of HNDOFF [g] (at line 14 or line 24)—call it (J)—must come after (G). Similarly, q_2's previous read of HNDOFF [g] before (I)—call it (K)—must come after (H).

We consider two cases. First, suppose (H) is an execution of line 18 by q_1. In this case, between (I) and (H), q_1 read POST [g] = v_{q1}, either at line 15 or at line 26; call this read (L). By the *Single Location Lemma*, because v_p is in p's set, the read at (L) implies that v_p was not guarded by g at (L). Therefore, v_p was not trapped by g at (L), which implies that it is safe for p to break out of the loop after (D) in this case (observe that $attempts_p = 2$ in this case).

For the second case, suppose (H) is an execution of line 29 by thread q_1. In this case, because q_1 is storing `null` instead of a value in its own set, the above argument does not work. However, because p breaks out of the loop at line 25 only if it reads a non-`null` value from HNDOFF [g] at line 24, it follows that if p does so, then some successful CAS stored a non-`null` value to HNDOFF [g] at or after (H), and in this case the above argument can be applied to that CAS to show that v_p was not trapped. If p reads `null` at line 24 after (D), then it continues through its next loop iteration.

In this case, there is a successful CAS (I) that comes after (H). Because (H) stored `null` in the current case, no subsequent execution of line 29 by any thread will succeed before the next successful execution of the CAS in line 18 by some thread. To see why, observe that the CAS at line 29 never succeeds while HNDOFF [g] contains `null` (see line 28). Therefore, for (I) to exist, there is a successful execution of the CAS at line 18 by some thread after (H) and at or before (I). Using this CAS, we can apply the same argument as before to conclude that v_p was not trapped. It is easy to see that PTB is wait-free.

As described so far, p picks up a value from HNDOFF [g] only if its value set contains a value that is guarded by guard g. Therefore, without some additional mechanism, a value stored in HNDOFF [g] might never be picked up from there. To avoid this problem, even if p does not need to remove a value from its set, it still picks up the previously handed off value (if any) by replacing it with null (see lines 28 through 30). We know it is safe to pick up this value by the argument above that explains why it is safe to pick up the value stored in HNDOFF [g] in line 18. Thus, if a value v is handed off to guard g, then the first `Liberate` operation to begin processing guard g after v is not trapped by g will ensure that v is picked up and taken to the next guard

(or returned from `Liberate` if *g* is the last guard), either by that `Liberate` operation or some concurrent `Liberate` operation.

Although various shared variables employed in the above exemplary realizations (e.g., `GUARDS[]`, `POST[]` and `HNDOFF[]`) are implemented as arrays of predetermined size, it is relatively straightforward to relax this restriction should it be desirable to do so in certain implementations or environments. For example, we could replace the `GUARDS` array by a linked list of elements, each containing at least one guard location. Association of posting and hand off locations with a given guard would be by any suitable data structure. Instead of stepping through the `GUARDS` array to hire a guard, threads would now traverse the linked list; if a thread reaches the end of the list without successfully hiring a guard, it can allocate a new node, and use CAS to attempt to atomically append the new node to the list. If this CAS fails, the thread resumes traversing the list from that point.

Other Embodiments

While the invention is described with reference to various implementations and exploitations, it will be understood that these embodiments are illustrative and that the scope of the invention(s) is not limited to them. Terms such as always, never, all, none, etc. are used herein to describe sets of consistent states presented by a given computational system, particularly in the context of correctness proofs. Of course, persons of ordinary skill in the art will recognize that certain transitory states may and do exist in physical implementations even if not presented by the computational system. Accordingly, such terms and invariants will be understood in the context of consistent states presented by a given computational system rather than as a requirement for precisely simultaneous effect of multiple state changes. This "hiding" of internal states is commonly referred to by calling the composite operation "atomic", and by allusion to a prohibition against any process seeing any of the internal states partially performed.

Many variations, modifications, additions, and improvements are possible. For example, while application to particular concurrent shared objects and particular implementations thereof have been described in detail herein, applications to other shared objects and other implementations will also be appreciated by persons of ordinary skill in the art. In addition, more complex shared object structures may be defined, which exploit the techniques described herein. Other synchronization constructs or primitives may be employed. For example, while many implementations have been described in the context of compare-and-swap (CAS) operations, based on that description, persons of ordinary skill in the art will appreciate suitable modifications to employ alternative constructs such as a load-linked, store-conditional (LL/SC) operation pair or transactional sequence or facilities of transactional memory, should such alternative constructs be available or desirable in another implementation or environment.

Plural instances may be provided for components, operations or structures described herein as a single instance. Finally, boundaries between various components, operations and data stores are somewhat arbitrary, and particular operations are illustrated in the context of specific illustrative configurations. Other allocations of functionality are envisioned and may fall within the scope of the invention(s). In general, structures and

functionality presented as separate components in the exemplary configurations may be implemented as a combined structure or component. Similarly, structures and functionality presented as a single component may be implemented as separate components. These and other variations, modifications, additions, and improvements may fall within the scope of the invention(s).

WHAT IS CLAIMED IS:

1. A method of supporting dynamic sizing for a shared data structure, the method comprising:
in a thread of a multithreaded computation, dereferencing a pointer read from the shared data structure after successfully posting a guard on a value that encodes the pointer; and
freeing storage referenced by the value only after storage referenced by the pointer is not in the shared data structure and no successfully posted guard thereon remains uncancelled.

2. The method of claim 1,
wherein the posting is successful if, after the posting, the storage referenced thereby remains in the shared data structure.

3. The method of any one of the preceding claims, further comprising:
in a second thread of the multithreaded computation, dereferencing a second pointer read from the shared data structure after successfully posting a guard on a second value that encodes the second pointer.

4. The method of claim 3,
wherein the value and the second value encode a same pointer.

5. The method of any one of the preceding claims, further comprising:
canceling the guard posted by the thread.

6. The method of any one of the preceding claims, further comprising:
qualifying for deallocation from the shared data structure, the storage referenced by the value.

7. The method of claim 6, further comprising:
returning storage referenced by the deallocation qualified value to a storage pool associated with the shared data structure.

8. The method of claim 7,
wherein the freeing is performed based on a storage pool management policy.

9. The method of claim 6, further comprising:
returning to an operating system, storage referenced by the deallocation qualified value.

10. The method of any one of the preceding claims, further comprising:
qualifying for deallocation from the shared data structure, storage referenced by a set of values, including the value.

11. The method of claim 1,
wherein the dereferencing and the freeing are performed by different threads of the multithreaded computation.

12. The method of claim 1,
wherein the dereferencing and the freeing are performed by a same thread of the multithreaded computation.

13. The method of any one of the preceding claims, further comprising:
dynamically allocating new storage; and
including a pointer thereto in the shared data structure.

14. The method of any one of the preceding claims,
wherein the shared data structure includes a collection of elements traversable, at least in part, by respective pointers; and
wherein the value upon which the guard is posted encodes one of the pointers.

15. The method of claim 14,
wherein the collection of elements is organized, at least in part, as a list; and
wherein the respective pointers include next pointers amongst elements of the list.

16. The method of any one of the preceding claims,
wherein the shared data structure implements a queue, stack, buffer, tree or list.

17. The method of claim 1, embodied, at least in part as:
a first instruction sequence that performs the guard posting and supplies the pointer for dereferencing use after success thereof.

18. A computer program product encoded in one or more computer readable media and including a dynamically-sizable implementation of a data structure comprising:
an representation of the data structure instantiable as separately reclaimable blocks of storage sharable, at least in part, by plural threads of a multithreaded computation; and
a first instruction sequence executable by one or more of the threads to access the data structure, the instruction sequence dereferencing a pointer read from the shared data structure only after successfully posting a guard on a value that encodes the pointer.

19. The computer program product of claim 18,
wherein the posting is successful if, thereafter, a respective one of the blocks referenced thereby remains in the shared data structure.

20. The computer program product of claim 18, further comprising:

a second instruction sequence executable by one or more of the threads to free storage referenced by the value only after the storage is not in the shared data structure and no successfully posted guard thereon remains uncancelled.

21. The computer program product of claim 18, further comprising:

a third instruction sequence executable by one or more of the threads to qualify for deallocation from the data structure a respective one of the storage blocks referenced by the value.

22. The computer program product of claim 18, further comprising:

a third instruction sequence executable by one or more of the threads to qualify for deallocation from the data structure respective ones of the storage blocks referenced by a set of values, including the value.

23. The computer program product of claim 21, further comprising:

a fourth instruction sequence executable by one or more of the threads to return storage referenced by the deallocation qualified value to a storage pool associated with the data structure.

24. The computer program product of claim 23,

wherein particular storage is freed from the storage pool only after no guard successfully posted on a value that references the particular storage remains uncancelled.

25. The computer program product of claim 21, further comprising:

a fifth instruction sequence executable by one or more of the threads to return storage referenced by the deallocation qualified value to an operating system.

26. The computer program product of claim 18,

embodied as a part of a memory allocator.

27. The computer program product of claim 18,

embodied, at least in part, as a garbage collection facility.

28. A computer program product encoded in one or more computer readable media and comprising:

an instruction sequence that includes one or more instructions that dereference a pointer of a data structure that is dynamically-sizable and shared amongst plural threads of a computation,

wherein, when executed, the instruction sequence posts, prior to the dereferencing, a guard on a value that encodes the pointer.

29. The computer program product of claim 28, further comprising:

an instantiable representation of the data structure.

30. The computer program product of claim 28, further comprising:

a second instruction sequence that frees storage referenced by the pointer after determining that the storage is not in the data structure and that no guard successfully posted on a value that encodes the pointer remains uncancelled.

31. A computer program product encoded in one or more computer readable media and comprising:

an instruction sequence that includes one or more instructions that free storage from a data structure that is dynamically-sizable and shared amongst plural threads of a computation,

wherein, when executed, the instruction sequence frees the storage after determining that the storage is not in the data structure and that no guard successfully posted on a value that encodes a pointer thereto remains uncancelled.

32. An apparatus comprising:

means for ensuring that, prior to dereferencing a pointer of a shared data structure, a thread posts a guard that traps the pointer; and

means for ensuring that prior to freeing storage referenced via the pointer, no thread of a multithread computation

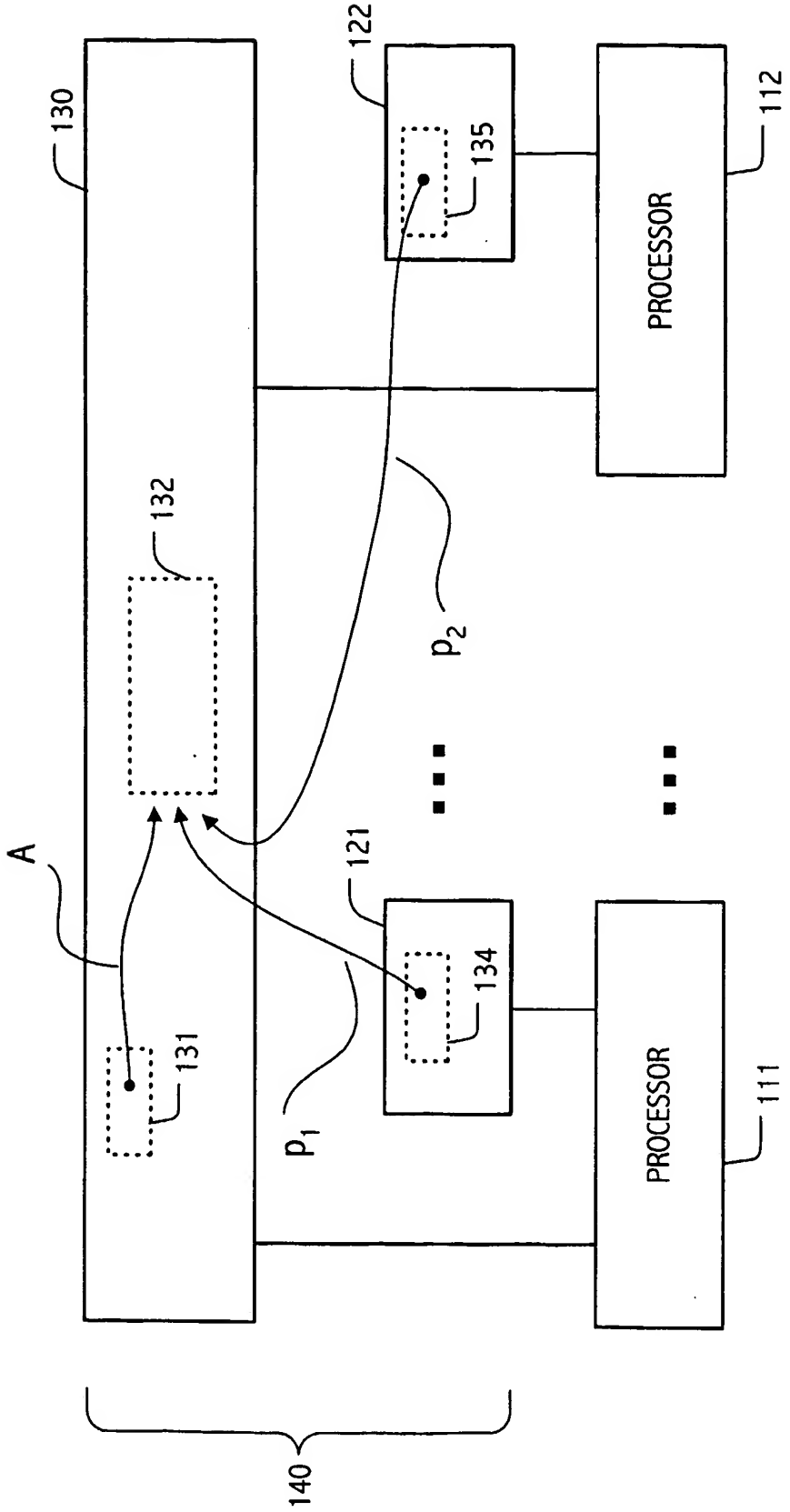

130
122
135
112
PROCESSOR
132
p_2
A
121
134
131
p_1
PROCESSOR
111
140

FIG. 1

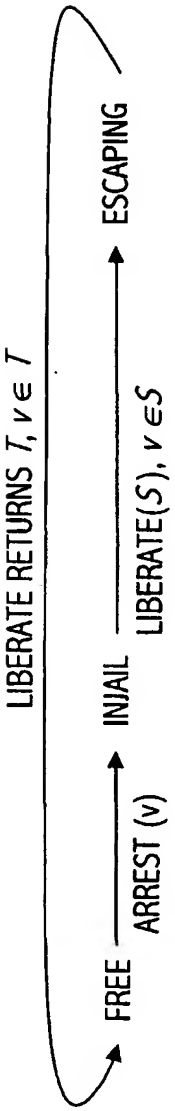

FREE
ARREST (v)
INJAIL
LIBERATE(S), v ∈S
ESCAPING
LIBERATE RETURNS T, v ∈ T

FIG. 2

actions

Environment	ROP output
$\mathsf{HireInv}_p()$	$\mathsf{HireResp}_p(g)$
$\mathsf{FireInv}_p(g)$	$\mathsf{FireResp}_p()$
$\mathsf{PostInv}_p(g,v)$	$\mathsf{PostResp}_p()$
$\mathsf{LiberateInv}_p(S)$	$\mathsf{LiberateResp}_p(S)$
$\mathsf{Arrest}(v)$	

state variables

For each client $p \in P$:
  pc_p: {idle, hire, fire, post(g,v), injail(v), liberate} **init** idle
  $guards_p$: set of guards **init** empty
For each value $v \in V$:
  $status[v]$: {injail, escaping, free} **init** free
For each guard $g \in G$:
  $post[g] : V$ **init null**;
  $trapping[g]$: **bool init** false;
$numescaping$: **int init** 0

transitions

$\mathsf{HireInv}_p()$
Pre: $pc_p = \mathsf{idle}$
Eff: $pc_p = \mathsf{hire}$

$\mathsf{FireInv}_p(g)$
Pre: $pc_p = \mathsf{idle}$
  $g \in guards_p$
  $post[g] = \mathbf{null}$
Eff: $pc_p = \mathsf{fire}$
  $guards_p = guards_p - \{g\}$

$\mathsf{PostInv}_p(g,v)$
Pre: $pc_p = \mathsf{idle}$
  $g \in guards_p$
Eff: $pc_p = \mathsf{post}(g,v)$
  $post[g] = \mathbf{null}$
  $trapping[g] = \mathsf{false}$

$\mathsf{LiberateInv}_p(S)$
Pre: $pc_p = \mathsf{idle}$
  for all $v \in S$,
    $v \neq \mathbf{null}$ and $status[v] = \mathsf{injail}$
Eff: $pc_p = \mathsf{liberate}$
  $numescaping = numescaping + |S|$
  for all $v \in S$, $status[v] = \mathsf{escaping}$

$\mathsf{Arrest}(v)$
Pre: $status[v] = \mathsf{free}$
  $v \neq \mathbf{null}$
Eff: $status[v] = \mathsf{injail}$
  for all g such that $post[g] = v$,
    $trapping[g] = \mathsf{true}$

$\mathsf{HireResp}_p(g)$
Pre: $pc_p = \mathsf{hire}$
  $g \in G$
  $g \notin \bigcup_q guards_q$
Eff: $pc_p = \mathsf{idle}$
  $guards_p = guards_p \cup \{g\}$

$\mathsf{FireResp}_p()$
Pre: $pc_p = \mathsf{fire}$
Eff: $pc_p = \mathsf{idle}$

$\mathsf{PostResp}_p()$
Pre: for some g, v, $pc_p = \mathsf{post}(g,v)$
Eff: $pc_p = \mathsf{idle}$
  $post[g] = v$
  $trapping[g] = (status[v] = \mathsf{injail})$

$\mathsf{LiberateResp}_p(S)$
Pre: $pc_p = \mathsf{liberate}$
  for all $v \in S$,
    $status[v] = \mathsf{escaping}$
    and for all $g \in \bigcup_q guards_q$,
      $(post[g] \neq v$ or $\neg trapping[g])$
Eff: $pc_p = \mathsf{idle}$
  $numescaping = numescaping - |S|$
  for all $v \in S$, $status[v] = \mathsf{free}$

FIG. 3

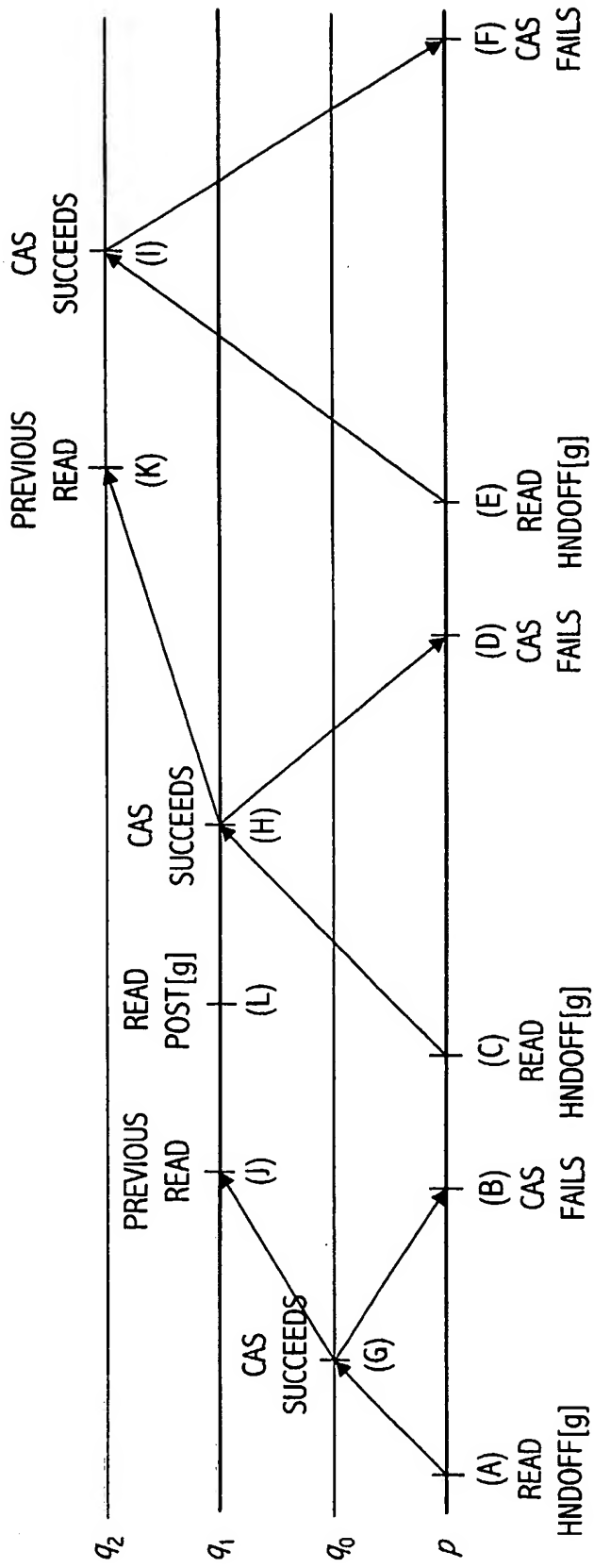

q_2
q_1
q_0
p
(A) READ HNDOFF[g]
(B) CAS FAILS
(C) READ HNDOFF[g]
(D) CAS FAILS
(E) READ HNDOFF[g]
(F) CAS FAILS
CAS SUCCEEDS (G)
PREVIOUS READ (I)
READ POST[g] (L)
CAS SUCCEEDS (H)
PREVIOUS READ (K)
CAS SUCCEEDS (I)

FIG. 4